IJCSI International Journal of Computer Science Issues, Vol. 4, No. 2, 2009
ISSN (Online): 1694-0784
ISSN (Print): 1694-0814

The Folklore of Sorting Algorithms

Dr. Santosh Khamitkar[1], Parag Bhalchandra[2], Sakharam Lokhande [2], Nilesh Deshmukh[2]

School of Computational Sciences,
Swami Ramanand Teerth Marathwada University,
Nanded (MS) 431605, India

Abstract

The objective of this paper is to review the folklore knowledge seen in research work devoted on synthesis, optimization, and effectiveness of various sorting algorithms. We will examine sorting algorithms in the folklore lines and try to discover the tradeoffs between folklore and theorems. Finally, the folklore knowledge on complexity values of the sorting algorithms will be considered, verified and subsequently converged in to theorems.

Key words: *Folklore, Algorithm analysis, Sorting algorithm, Computational Complexity notations.*

1. Introduction

Folklore is the traditional beliefs, legend and customs, current among people. Where as a theorem is a general conclusion which has been proved [1].In view of these definitions one might be tempted to conclude simply that the folk theorem is a general conclusion which is traditional and can be proved. The objective of this paper is to narrate the folklore on complexity issues of sorting algorithms and prove them. Accordingly, we shall attempt to provide a reasonable definition for complexity or criteria for folklore, followed by a detailed example illustrating ideas. The letter endeavor might take one of the two possible forms. We could take a piece of folklore and show that it is a theorem or take a theorem and show that it is folklore. Literature review in present context highlights that the unified theory regarding their folklore knowledge and in terms of theorems is found missing.

Since the dawn of computing, the sorting problem has attracted a great deal of research. Till date, Sorting algorithms are open problems and many researchers in past have attempted to optimize them with optimal space and time scale. Here Optimization was thought as the process to reduce the time complexity so as to cross at least O (n log n) milestone. Many new techniques were proposed and till today fine refinement of them is in progress. Every researcher attempted optimization in past has found stuck to his / her observations regarding sorting experiments and produced his/ her own results. Since these results were specific to software and hardware environment therein, we found abundance in the complexity analysis which possibly could be thought as the folklore knowledge. We tried to review this folklore knowledge in past research work and came to a conclusion that it was mainly devoted on synthesis, optimization, effectiveness in working styles of various sorting algorithms.

Our work is not mere commenting earlier work rather we are putting the knowledge embedded in sorting folklore with some different and interesting view so that students, teachers and researchers can easily understand them. Synthesis, in terms of folklore and theorems, hereupon aims to check what past researchers have thought is really the same they were saying and to analyze research on sorting algorithms in such a way to get united understanding of and support for the research results so that they can be used directly and defended in public. Since we have resolved to introduce no new technical material in this paper except having technical elaboration of folklore, and as researchers seem to be less familiar with folklore than with theorems, we prefer to deal with both approaches stated in above Para. We will present strong evidence to the effect that a particular folk can be converged in to theorem and vice versa. Folklore knowledge on complexity values of the sorting algorithms will be considered, verified and subsequently converged in to theorems.

This paper is important as Sorting algorithms are often prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts. Sorting algorithms illustrate a number of important principles of algorithm design; some of them are also counterintuitive [2]. Efficient sorting is important to optimizing the use of other algorithms such as Binary search and merge algorithms that require sorted lists to work correctly [2,3]. We can not deploy binary search if data is not pre sorted otherwise the search process may get trapped into a blind alley thereby exhibiting worst case complexity. Literature review has highlighted mainly interesting things which could be the parts of folklore and or theorems .Usually these folklore and theorem parts are always omitted by the faculties teaching these topics and students find them extremely difficult to understand. We will examine sorting algorithms in these lines and try and discover the tradeoffs between folklore and theorems. This paper covers different complexity aspects of basic sorting models, such as big O notation, best, worst and average case analysis, time-space tradeoffs, lower bounds, etc.

2. **Origin of Folklore**

When we look to develop or use a sorting algorithm on large problems, it is important to understand how long the algorithm might take to run. The time for most sorting algorithms depends on the amount of data or size of the problem. In order to analyze an algorithm, we try to find a relationship showing how the time needed for the algorithm depends on the amount of data. This is called the "complexity" of the algorithm. [4]

A simple sorting algorithm like bubble sort may have a high complexity, whereas an algorithm which is very complex in its organization such as shell sort , sometimes pays off by having a lower complexity in the sense of time needed for computation[5,6]. Besides running time analysis, it is seen that, factors other than the sorting algorithm selected to solve a problem, affect the time needed for a program. It is just because different people carrying out a solution to a problem may work at different speeds, even when they use the same sorting method, different computers work at different speeds. The different speeds of computers on the same program can be due to different "clock speeds", the rates at which steps in the program are executed by the machine and different "architectures," the way in which the internal instructions and circuitry of the computer are organized. Literature review shows that, the researchers have attempted for rigorous analysis of sorting algorithms and produced prolific comments [7, 8, 9] discovering such complexity dependent factors .

Consequently, analysis of sorting algorithm can not predict exactly how long it will take on a particular computer [5]. What analysis can do is tell us how the time needed depends on the amount of data. For example, we always come across folklore like, for an algorithm, when we double the amount of data, the time needed is also doubled, and in other words the time needed is proportional to the amount of data. The analysis of another algorithm might tell us that when we double the amount of data, the time is increased by a factor of four, which is the time needed is proportional to the amount of data squared. The latter algorithm would have the time needed increase much more rapidly than the first.

When analyzing sorting algorithms, it often happens that the analysis of efficiency also depends considerably on the nature of the data. For example, if the original data set already is almost ordered, a sorting algorithm may behave rather differently than if the data set originally contains random data or is ordered in the reverse direction. Similarly, for many sorting algorithms, it is difficult to analyze the average-case complexity. Generally speaking, the difficulty comes from the fact that one has to analyze the time complexity for all inputs of a given length and then compute the average. This is a difficult task. Using the incompressibility method, we can choose just one input as a representative input. Via Kolmogorov complexity, we can show that the time complexity of this input is in fact the average-case complexity of all inputs of this length. Constructing such a “representative input” is impossible, but we know it exists and this is sufficient [10]

For these reasons, the analysis for sorting algorithms often considers separate cases, depending on the original nature of the data. The price of this generality is exponential complexity; with the result that many problems of practical interest are solvable better than folklore of sorting, but the limitations of computational capacity prevent them from being solved in practice. The increasing diversity in computing platforms motivates consideration of multi-processor environment. Literature review suggests that no folklore is found mentioned regarding complexity in multiprocessor environment.

Recently, many results on the computational complexity of sorting algorithms were obtained using Kolmogorov complexity (the incompressibility method). Especially, the usually hard average-case analysis is amenable to this method. A survey [10] shows such results about Bubble sort, Heap sort, Shell sort, Dobosiewicz-sort, Shaker sort, and sorting with stacks and queues in sequential or parallel mode. It is also found that the trade-off between memory and sorting is enhanced by the increase in availability of computer memory and the increase in processor speed. Currently, the prices of computer memory are decreasing. Therefore, acquiring larger memory configurations is no longer an obstacle, making it easier to equip a computer with more memory.

Similarly, every year there is an increase in computer speed. The Moore’s Law, a computer-industry standard stated that computer chips should double in power approximately every 18 months. The speed of most of today's computers is really quite remarkable. Even the "slower" ones are quite fast, and the actual processing doesn't take much time. Delays, usually encountered in "I/O", either to/from disk, the network or peripheral devices have been eliminated because of innovative interfacing standards, thereby boosting speed. If a computer has an 800 MHz processor, sounds like it was made in the early to mid 90's. If we look at most computers coming out today we will notice that the dual core processor - which is in fact two processors on the computer, each processor is clocking in at around 1.85 Ghz, which alone is more than two times as fast as that 800 Mhz processor. Then we have to multiply that by two since there are two cores. It is something like today's computers are working about 4-5 times faster than the earlier ones. Such increase in computer speed causes acceleration of sorting algorithms. Thus knowledge proved by some researcher for a sorting algorithm on its complexity can not be considered absolute as it may be increased or decreased analogously. Therefore piece of knowledge seen can be thought as folklore knowledge.

Above discussion is potentially the source for folklore. Many people, same work on different environment leading to different folklores. Some examples of folklore knowledge are 1) A quick sort is an effective, but more complex, sorting algorithm, 2) Shell Sort is much easier to code than Quicksort, and it is nearly as fast as Quicksort.

In next part of paper, we try to express sorting algorithm's behavior using the folklore and then try to prove it by formulating theorems.

3. Background Knowledge

Sorting is always carried out technically. In computer science and mathematics; we can formulate a procedure for sorting unordered array or a file. Such procedure is always governed by an algorithm; called Sorting Algorithm. It is highly expected that we must converge to the definition of sorting concept. Basically a sorting problem is to arrange a sequence of objects so that the values of their "key" fields are in "non-decreasing" sequence.
That is, given objects

$$O_1, O_2, ..., O_n$$

With key values

$$K_1,\ K_2, ...,\quad K_n \quad \text{respectively,}$$

Arrange the objects in an order

$$O_{i1},\ O_{i2},\quad ...,\ O_{in}\ ,$$

Such that

$$K_{i1} <= K_{i2} \quad <= ... <= Kin. \qquad (1)$$

Generally, Computational complexity (worst, average and best behavior) of element comparisons in terms of the size of the unsorted list is considered for the analysis of the efficiency of sorting algorithm. The complexity notational terminology is covered in [2]. If the size of unsorted list is (n), then for typical sorting algorithms, good behavior is O (n log n) and bad behavior is Ω (n²). The Ideal behavior is O (*n*). Sort algorithms which only use an abstract key comparison operation always need Ω (*n* log *n*) comparisons in the worst case.
An important criterion used to rate sorting algorithms is their running time. Running time is measured by the number of computational steps it takes the sorting algorithm to terminate when sorting n records. We say that an algorithm is O (n^2) ("of order *n*-squared") if the number of computational steps needed to terminate as *n* tends to infinity increases in proportion to $n^{2.}$ Literature review carried out in [11, 5] indicate the man's longing efforts to improve running time of sorting algorithm with respect to above core algorithm concepts.

Research on efficiency analysis [3, 2] use big O, omega and theta notation to give asymptotic upper, lower, and tight bounds on time and space complexity of sorting algorithms. They determine the time complexity of sorting algorithms, use a list of functions and order them according to asymptotic growth.

4. Some Folklore and Convergence in to Theorems

4.1 Folklore 1: The running time of a sorting algorithm grows if the length of the input sequence grows.

Proof 1: Proof can be obtained from the following reformulation of the sorting problem. For *n* records there are *n*! possible linear arrangements, only one of which is the correct, sorted set of records. (*n*! = 1 x 2 x 3 x . . . [*n* - 2] x [*n* - 1] x *n*.) One may imagine these *n*! arrangements as the leaves of a binary decision tree. The process of sorting then involves starting at the root of the tree (the unordered initial list) and traveling down the nodes, choosing either the left or right node based on results of comparisons. (In this terminology, the "root" of a tree is at the top, and all the branches expand downward.) The maximum number of steps needed would correspond to the height of the tree, which is log *n*!. Using Stirling's approximation of log *n*!. For large *n*, this process would be O (*n* log *n*).

Thus the folklore 1 can be converged in to a theorem as, **Theorem 1: For n records, linear sorting will take a complexity O (n log n).** If the input sequence is presorted, compared to an unsorted sequence possibly less steps are sufficient.

Seldom, the theorem does not hold true. It might be the case that the keys of the objects fall within a known range. Let there be a controlled environment, where we produce the keys. For example, we might issue employee numbers in a given order and in a given sequence, or we might issue part numbers to items that we manufacture. Therefore, if the keys are unique and known to be in a given range, we might allow the key to be the location of the object (imagine placing the objects in a vector such that object with key K is in (vector-ref v k)). However the algorithms presented only rely on the fact that we can compare keys, to determine if one is less than another

4.2 Folklore 2: In general, we can't sort faster than O (n log n).

Proof 2: Consider Heap Sort or Bubble Sort or Quick Sort or Merge Sort or anything else like those, whether already invented or still to be invented. They all will take at least *O (n log n)* steps [5] to sort *n* items, because of the reason that, they all work by comparing data items and then moving data items around based on the outcome of the comparisons [5]. To illustrate this, here is a piece of Heap Sort:

If (a[c/2] < a[c]) // if parent is smaller than child

Swap (c/2, c); // trade their values

Imagine running such a sort on these numbers:

3, 2, 4, 5, 1

The sort will compare the numbers and based on the outcomes of that comparison move the values around, with the net effect of all the moves being summarized by these arrows:

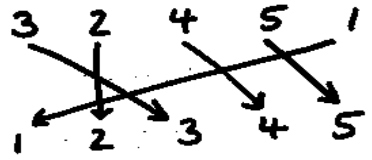

Fig.1. Suggested moves during swapping process

However it cannot run the same way on same but differently arranged inputs like 2, 3, 4, 5, 1, as output will be misleading. Thus the execution must take a different path for every permutation of inputs.

We can visualize all these executions as an execution tree. Each node is some action, either an assignment or a comparison; the tree branches after each comparison; each path from the root to a leaf is one execution of the program; and the height of the tree is the (worst-case) running time of the program. If this is a sorting program that sorts the numbers 1 through *n* then execution must reach a different leaf for each of the *n!* permutations of 1....*n*.

We know that a binary tree with *L* leaves has height at least *log L* where the *log* is base 2. Thus our execution tree has height at least *log (n!)* which is *O (n log n)*.To conclude, these sorting algorithms must take a different path of execution for every different permutation of the input values. There are lots of different permutations (*n!*). To be able to run in that many different ways (i.e., for the execution tree to have that many leaves), the programs cannot run too quickly (i.e., the execution tree cannot be too shallow). We may want to conclude that sorting cannot be done in less than *O (n log n)* time. This would be an overstatement because there is a subtle assumption in the above argument: that the program determines what actions to take (i.e., how to move values around) solely based on the outcome of comparisons between the input values. The lesson learned is, programs that *base their actions only on comparisons of the data* cannot beat the *O (n log n)* barrier. To have a chance at sorting faster we have to avoid comparing data! , which is highly impossible. However, there are situation where we can say that we can sort faster than O (n log n). For example, Counting Sort and Radix Sort work faster when the range of values is limited .Thus the folklore 2 can be converged in to a theorem as, **Theorem 2: For n records, sorting will take minimum O (n log n) time.**

4.3 Folklore 3: The exact number of steps required by insertion sort is given by the number of inversions of the sequence

Proof 3: Let $a = a_0, ..., a_{n-1}$ be a finite sequence. An inversion is a pair of index positions, where the elements of the sequence are out of order. Formally, an inversion is a pair (i, j), where $i < j$ and $a_i > a_j$. Example: Let $a = 5, 7, 4, 9, 7$. Then, (0, 2) is an inversion, since $a_0 > a_2$, namely $5 > 4$. Also, (1, 2) is an inversion, since $7 > 4$, and (3, 4) is an inversion, since $9 > 7$. There are no other inversions in this sequence. The inversion sequence $v = v_0, ..,v_{n-1}$ of a sequence $a = a_0, ..., a_{n-1}$ is defined by

$v_j = |\{ (i, j) \mid i < j \wedge a_i > a_j \}|$ for $j = 0, ..., n-1$.

The above sequence $a = 5, 7, 4, 9, 7$ has the inversion sequence $v = 0, 0, 2, 0, 1$.

We now count the inversions (i, j) of a sequence a separately for each position j. The resulting value of v_j gives the number of elements a_i to the left of a_j which are greater than a_j. Obviously, we have $v_i \leq i$ for all $i = 0, ..., n-1$. If all v_i are 0, then and only then the sequence a is sorted. If the sequence a is a permutation, then it is uniquely determined by its inversion sequence v. The permutation $n-1, ..., 0$ has inversion sequence $0, ..., n-1$.

Let $a = a_0, ..., a_{n-1}$ be a sequence and $v = v_0, ..., v_{n-1}$ be its inversion sequence. Then the number of steps $T(a)$ required by insertion sort to sort sequence a is

$$T(a) = \sum_{i = 0, ..., n-1} v_i \qquad (2)$$

Obviously, insertion sort takes v_i steps in each iteration i. Thus, the overall number of steps is equal to the sum of all v_i. This can be proved by an example: The following table shows the sequence a of the first example and its corresponding inversion sequence. For instance, $v_5 = 4$ since 4 elements to the left of $a_5 = 2$ are greater than 2 (namely 5, 7, 3 and 4). The total number of inversions is 17.

Table 1: Inversion Table

i	0	1	2	3	4	5	6	7
a_i	5	7	0	3	4	2	6	1
v_i	0	0	2	2	2	4	1	6

It is seen that insertion sort requires 17 steps to sort this sequence.Thus the folklore 3 can be converged in to a theorem as, **Theorem 3 : The sorting algorithm insertion sort has a worst case time complexity of $T(n) = n \cdot (n-1)/2$ comparison-exchange steps to sort a sequence of n data elements.**

4.4 Folklore 4: Any comparison sort type of algorithm requires lower bound comparisons in the worst case.

Proof 4: From the folklore 2, it suffices to determine the height of a decision tree in which each permutation appears as a leaf. Consider a decision three of height h and l leaves corresponding to a comparison sort on n elements. Because each of the n! permutations of the input appear as some leaf, we have $n! \leq l$.

Since a binary tree of height h has no more than 2h leaves, we have to conclude that $n! \leq l \leq 2$, which by taking logarithms , implies $h \geq \log(n!)$ (Since the log function is monotonically increasing).
Therefore,

$$\begin{aligned}
h &\geq \log(n!) \\
&= \log(n.n1.n2....2.1) \\
&= \log n + \log(n1) + \log(n2) + ... + \log 2 + \log 1 \\
&> \log n + \log(n1) + \log(n2) + ... + \log(n/2) \\
&> \log(n/2) + \log(n/2) + ... + \log(n/2) \\
&= n/2 \log(n/2) \\
&= n/2 \log n - n/2 \\
&= \Omega(n \log n) \qquad (3)
\end{aligned}$$

This folklore can be converged in to new theorem as, **Theorem 4: any comparison sort algorithm requires Ω (n log n) comparisons in the worst case.**

5. Some Convergence of Theorems into Folklore

Now we take a theorem and show that it is a piece of folklore. We know that time needed for these algorithms like selection sort varies like N^2. [12]. Let's deduce folklore form this.

5.1 Deduction 1:

Algorithms like selection sort, takes each unsorted list and goes through each item in order to identify the smallest. The first unsorted list consists of all N items in the original, and each successive unsorted list is one smaller, as the smallest item each time is placed in its correct position. So the first search for the smallest will take N operations, the next search for the smallest will take N-1 operations, the next N-2 operations, and so on until the last search for the smallest take 2 operations (we don't need to search when the list has only one item). Thus the time needed will be a constant times the following sum:

$$N + (N-1) + (N-2) + ... + 3 + 2 + 1 \qquad (4)$$

(We include the 1 to create a sum for which a standard formula applies; it adds only a small constant which does not affect the overall result.)

This sum, the sum of the first N integers, is known to be N (N+1) / 2, which is approximately $N^2/2$. Since we know that this is multiplied by some constant representing the time the operations take on a specific machine, we can see that the time needed for this algorithm varies like N^2, the square of the amount of data. Thus the time needed for the selection sort algorithm behaves like; each time the size of the list is doubled, the amount of time needed is increased by a factor of four. **The folklore deduced is "Algorithms like Selection sort are quadratic in nature."**

6. Conclusion

In this paper we have shown that the research on sorting algorithm has produced abundant results and can be thought as Folklore knowledge - momentarily belief among people. We proved that these results are not absolute as they are specific to some factors like person working, configuration of hardware, programming styles, whether parallelism is opted or not? Etc. Even with these uncertainties, we have analyzed sorting algorithms and presented strong evidence to the effect that a particular folk can be converged in to theorem and vice versa so as to show that folklore and converged theorems are interchangeable**.**

References

[1] Oxford Dictionary

[2]Horowitz, E.,Sahni. S, **Fundamentals of Computer Algorithms,** Computer Science Press, Rockville. Md.

[3] Dr. D. E. Knuth, The Art of Computer Programming, 3rd volume, "**Sorting and Searching**", second edition.

[4] Liu C. L., "Analysis of sorting algorithms", Proceedings of Switching and Automata Theory, **12th Annual Symposium, 1971**, East Lansing, MI, USA, pp 207-215.

[5] Darlington. J, A synthesis of several sorting algorithms, **Acta Inf. II,** 1978, pp 1-30.

[6] John Darlington, Remarks on "A Synthesis of Several Sorting Algorithms", **Springer Berlin / Heidelberg**, pp 225-227, Volume 13, Number 3 / March, 1980.

[7] Talk presented by Dr. Sedgewick ," Open problems in the analysis of sorting and searching algorithms", at the **Workshop on the probabilistic analysis of algorithms**, Princeton, May, 1997.

[8] A. Andersson & T. Hagerup, S. Nilsson, and R. Raman, "Sorting In Linear Time?" **Proceedings of the 27th Annual ACM Symposium on the Theory of Computing**, 1995.

[9] Parag Bhalchandra, "Proliferation of Analysis of Algorithms with application to Sorting Algorithms", **M.Phil Dissertation**, VMRF, India, July 2009

[10] Paul Vitanyi , "Analysis of Sorting Algorithms by Kolmogorov Complexity (A Survey) ", appeared **in Entropy, Search, Complexity, Bolyai Society Mathematical Studies**, pp 209—232 , Springer-Verlag, 2007

[11] Richard Harter, Inefficient sort algorithms – "A Computer Environment for Beginners' Learning of Sorting Algorithms: Design and Pilot Evaluation", ERIC**, Journal Number 795978, Computers & Education**, v51 n2, pp708-723, Sep 2008

[12] S Nilsson, "the Fastest Sorting Algorithm?" **Doctor Dobbs Journal,** 2000.

Dr. S. D. Khamitkar, M.Sc. PhD
He is Associate Professor and has more than 14 years of teaching and research experience. He has published near abut 6+ papers in international journals. He is research guide and currently ten students are working with him His interest area includes Network Security, Scientific computing, etc. He is life member of Indian Science Congress Association, ISTE.

Parag Bhalchandra, Sakharam Lokhande, Nilesh Deshmukh, M.Sc., SET-NET, M.Phil
All of them are Assistant Professors and have registered for PhD research work. They have 8+ years teaching experience and have 2+ papers in international conferences. They are life members of ISCA, ISTE. Their interest lies in Algorithm Analysis, soft computing and web based developments.